Technical Report
Mini UNIX Shell Implementation

Gaurav Jain (2025MCS2121)
Aman Singh (2025MCS2122)

December 9, 2025

Abstract

This report documents the design and implementation of the Mini UNIX Shell, a custom command-line interpreter developed in C. The system implements a Read-Eval-Print Loop (REPL) capable of tokenizing user input, managing process execution via system calls (fork, execvp), handling I/O redirection, and orchestrating inter-process communication using pipes.

1 System Architecture

The shell is modularized into four primary components, ensuring separation of concerns:

- **Main Controller** (`main.c`): Manages the REPL loop and signal handling.
- **Parser** (`parser.c`): Tokenizes input and builds command structures.
- **Executor** (`executor.c`): Handles process creation and file descriptor manipulation.
- **Builtins** (`builtins.c`): Implements internal shell commands.

2 Data Structures

The core data structure driving the shell is the `command_t` struct, defined in `parser.h`. It utilizes a linked-list approach to support pipelining:

```
typedef struct command
{
  char **argv;              // Arguments array
  char *input_redirect;     // File for '<' redirection
  char *output_redirect;    // File for '>' redirection
  int background;           // Background execution flag
  int is_exec;              // 1 = external, 0 = builtin
  struct command *pipe_to; // Pointer to next command in pipeline
} command_t;
```

3 Implementation Details

3.1 Parsing Logic

The parser processes input in two stages:

1. **Tokenization**: The `tokenize` function splits the raw input string into an array of tokens, handling whitespace and special characters like `|`, `<`, `>`, and `&` [**?**].

2. **Command Construction**: The `parse_tokens` function iterates through the tokens. Upon encountering a pipe (`|`), it finalizes the current command node and allocates a new one linked via the `pipe_to` pointer [**?**].

3.2 Process Execution

The executor distinguishes between simple commands and pipelines:

3.2.1 Simple Commands

For non-piped commands, the shell calls `fork()`. The child process sets up redirection using `dup2()` if necessary, and then replaces its image using `execvp()`. If the `background` flag is set, the parent does not `wait()` immediately; otherwise, it waits for the child to terminate [**?**].

3.2.2 Pipeline Management

Pipeline execution is handled by `execute_pipeline`. It iterates through the `command_t` list:

- A pipe is created using `pipe()` for every command (except the last).
- **Child Process**: Binds `STDIN` to the read-end of the previous pipe and `STDOUT` to the write-end of the current pipe using `dup2`.
- **Parent Process**: Closes the pipe ends it no longer needs to prevent file descriptor leaks and hangs [**?**].

3.3 Signal Handling

To provide a robust user experience, the shell modifies default signal behaviors:

- **SIGINT (Ctrl+C)**: Ignored in the parent process (`SIG_IGN`) to prevent the shell from exiting when the user interrupts a subprocess [**?**].
- **SIGCHLD**: A handler reaps zombie processes using `waitpid(-1, NULL, WNOHANG)`. This is critical for cleaning up background processes that finish asynchronously [**?**].

4 Conclusion

The Mini UNIX Shell successfully implements core OS concepts including process management, I/O redirection, and IPC. The use of a linked-list structure for commands allows for flexible pipeline construction, while dedicated signal handlers ensure stability during execution.

5 Appendix

Commit History

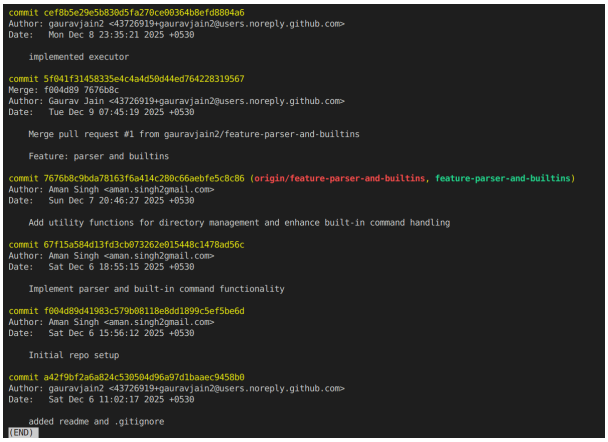

Figure 1: Commit History 1

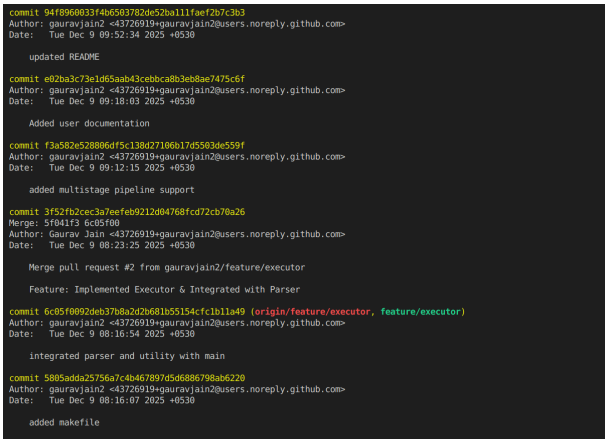

Figure 2: Commit History 2